Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementando Diversidade em Replicação Máquina de Estados

Caio Yuri da Silva Costa

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Eduardo Adilio Pelinson Alchieri

Brasília
2016

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Rodrigo Bonifácio de Almeida

Banca examinadora composta por:

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador) — CIC/UnB
Prof. Dr. Edison Ishikawa — CIC/UnB
Prof. Me. Marcos Fagundes Caetano — CIC/UnB

CIP — Catalogação Internacional na Publicação

Costa, Caio Yuri da Silva.

Implementando Diversidade em Replicação Máquina de Estados / Caio Yuri da Silva Costa. Brasília : UnB, 2016.
123 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2016.

1. Replicação Máquina de Estados, 2. tolerância a falhas bizantinas, 3. diversidade, 4. segurança

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementando Diversidade em Replicação Máquina de Estados

Caio Yuri da Silva Costa

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador)
CIC/UnB

Prof. Dr. Edison Ishikawa
CIC/UnB

Prof. Me. Marcos Fagundes Caetano
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordenador do Bacharelado em Ciência da Computação

Brasília, 1 de julho de 2016

Dedicatória

Dedico aos meus pais, meus eternos incentivadores, e aos meus amigos, que tanto torceram pelo meu sucesso na realização deste.

Agradecimentos

Agradeço ao Professor orientador Eduardo Alchieri, pela oportunidade e pela cobrança durante a execução deste projeto, e à família e amigos, por todo o suporte dado durante sua execução.

Resumo

Vulnerabilidades podem comprometer as propriedades de segurança de um sistema quando adequadamente exploradas por um adversário. Uma alternativa para mitigar este risco é a implementação de sistemas tolerantes a intrusões. Uma abordagem muito utilizada para estas implementações é a Replicação Máquina de Estados (RME). Porém, as soluções existentes não suportam diversidade na implementação das réplicas, de forma que um mesmo ataque pode comprometer todo o sistema. Neste sentido, este trabalho propõe uma arquitetura para fornecer suporte à diversidade de implementação em RMEs, através da utilização de múltiplas linguagens de programação, integradas ao Bft-SMaRt, que é uma biblioteca para implentar RME, por meio de uma camada intermediária em linguagem C. Um conjunto de experimentos mostra a viabilidade prática das soluções propostas, avaliando a performance e a segurança, através da implementação de uma aplicação que utiliza as funcionalidades desenvolvidas, além de explorar outros eixos de diversidade como: sistema operacional, hardware, compiladores e ambientes de execução.

Palavras-chave: Replicação Máquina de Estados, tolerância a falhas bizantinas, diversidade, segurança

Abstract

The security properties of a system could be impaired by an attacker that exploits its vulnerabilities. An alternative to mitigate this risk is the implementation of intrusion-tolerant systems. State Machine Replication (SMR) is widely used in these implementations. However, the proposed solutions do not allow diversity in replica implementations and consequently, the same attack can compromise the entirety of the system. In that context, this work proposes an architecture to allow diversity in replica implementations of a SMR, by employing multiple programming languages integrated into BFT-SMaRt, a library to implement SMR, by means of an intermediate layer written in C. A set of experiments shows the practical viability of the proposed solutions, evaluating performance and security, by implementing an application that uses the developed functionalities, while also exploring other diversity aspects like: operating system, hardware, compilers and runtime environments.

Keywords: State Machine Replication, byzantine fault-tolerance, diversity, security

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Neste capítulo são apresentadas a motivação e objetivos deste trabalho, além de descrito como o texto é organizado.

1.1 Motivação

Sistemas computacionais confiáveis devem garantir seu correto funcionamento mesmo com a falha de um ou mais de seus componentes. Funcionar corretamente significa manter suas propriedades, como disponibilidade, integridade e confidencialidade [4]. Para tolerar falhas é comum a implementação de uma ou mais soluções CFT (*crash fault-tolerance*). No entanto, tal medida é suficiente apenas para resistir à falha total de um componente—quando o mesmo se torna indisponível, deixando de se comunicar com o resto do sistema, e não oferece proteção contra um componente que passa a responder de forma maliciosa. Para garantir o funcionamento correto do sistema nessas situações, deve-se utilizar uma abordagem tolerante a falhas bizantinas [14].

Uma abordagem amplamente utilizada na implementação de sistemas tolerantes a falhas, tanto por *crash* [34] quanto bizantinas (maliciosas) [9] é a Replicação Máquina de Estados (RME) [34]: esta abordagem consiste em replicar os servidores (permitindo que alguns deles falhem, i.e., sejam invadidos e controlados por um adversário ou apresentem algum *bug* de *software* ou *hardware*) e coordenar as interações entre os clientes e as réplicas dos servidores, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados.

Embora existam muitas propostas de sistemas e protocolos para RME [3, 8, 9, 12, 13, 20, 23, 38], nenhum deles fornece suporte para implementação de diversidade nas réplicas [7, 17, 18, 28, 30], implementando diferentes réplicas em diferentes linguagens de programação aumentando o grau de independência de falhas (duas ou mais réplicas

não falham pelo mesmo motivo [27]). Sendo assim, nestes sistemas existentes, um adversário pode utilizar o mesmo ataque para comprometer todas as réplicas do sistema ou ainda, como o mesmo *software* está executando nas diversas réplicas, um mesmo *bug* pode comprometer toda a aplicação.

1.2 Objetivos

1.2.1 Geral

Com o objetivo de preencher a lacuna no suporte a diversidade em sistemas para RME, este trabalho apresenta nossos esforços para adicionar suporte a diversidade na implementação das réplicas no BFT-SMaRt [8]. O Bft-SMaRt é uma biblioteca escrita em Java para implementação de uma RME tolerante a falhas bizantinas. Nosso objetivo não é implementar diversidade do BFT-SMaRt em si, reimplementando-o em diferentes linguagens—mas sim fornecer interfaces para que as réplicas sejam escritas em linguagens de programação diversas, mantendo a implementação do BFT-SMaRt em Java. A arquitetura proposta utiliza o conceito de estado abstrato independente da linguagem [10] e apresenta pelo menos dois grandes desafios: *(1)* internamente em uma réplica – possibilitar a comunicação entre diferentes linguagens (ex.: uma réplica escrita em C precisa executar métodos do BFT-SMaRt escritos em Java e vice-versa); e *(2)* entre réplicas (ou entre réplicas e clientes) – possibilitar a troca de informações entre réplicas implementadas em linguagens diferentes (ex.: a representação de um vetor em C é diferente da em Java).

1.2.2 Específicos

Os objetivos específicos deste trabalho são as seguintes:

1. Estudar os conceitos envolvendo diversidade e RME.
2. Proposta de uma arquitetura para suportar diversidade na implementação de réplicas de uma RME, que seja extensível (*i. e.* possibilite a implementação de suporte a novas linguagens de programação) e que apresente baixa perda de performance devido à comunicação entre diferentes ambientes de execução.
3. Integração e descrição de como esta arquitetura foi integrada no Bft-SMaRt.

4. Apresentação e análise de uma série de experimentos com as implementações realizadas, trazendo uma melhor compreensão a respeito do funcionamento de diversidade em uma RME.

1.3 Organização do Texto

O texto deste trabalho está dividido em 5 capítulos. Neste primeiro, foram introduzidas a motivação e objetivos. No segundo capítulo, é fornecida uma introdução teórica acerca do tema abordado e das tecnologias utilizadas na implementação dos objetivos. No Capítulo 3 é detalhado como foi feita a implementação. No Capítulo 4 são apresentados os experimentos e seus resultados, e por fim no Capítulo 5 são apresentadas as conclusões deste trabalho.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta os conceitos básicos de segurança e falhas em sistemas distribuídos e diversidade, os quais serviram de base para a motivação deste trabalho, o BFT-SMaRt, plataforma que serviu de base para a implementação prática e por fim as tecnologias utilizadas que possibilitaram a implementação do trabalho.

2.1 Segurança de Funcionamento em Sistemas Distribuídos

Nesta seção serão introduzidos conceitos de falhas e segurança em sistemas distribuídos, e alternativas comuns para contorná-las.

2.1.1 Conceitos Básicos

Defeitos em *software* e sistemas acontecem quando um sistema se comporta de maneira inesperada, não atendendo à sua especificação. Um erro é a parte do estado do sistema responsável pelo defeito, que por sua vez, é causado por uma falha [25].

Quanto à sua natureza, as falhas podem ser acidentais ou intencionais; quanto à sua causa, podem ser humanas (*e.g.* *bugs* no código) ou físicas (*e.g.* problemas de *hardware*); quanto a escopo, podem ser internas, como causadas por um estado inconsistente, ou externas, como as causadas por interferência eletromagnética; e a depender do momento que foram introduzidas podem ser de *design*, ou operacionais (por exemplo, as que surgem por uma intrusão) [25].

Dada a diversa gama de fontes de falha em um sistema, é desejável que um sistema seja capaz de funcionar corretamente ainda que uma falha aconteça, *i.e.* é desejável que as falhas não gerem erros. Para tal, o sistema deve ser tolerante a falhas.

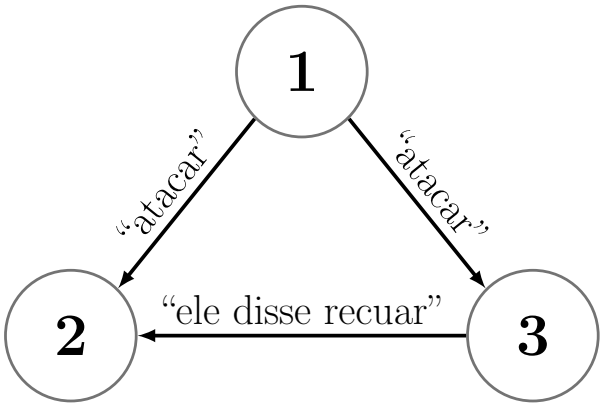

Figura 2.1: Situação de confusão causada por um traidor. Adaptada de Lamport [24]

A implementação mais comum de sistemas tolerantes a falhas é a CFT (*crash-fault tolerance*). Tal classe de falhas consiste na indisponibilidade ou queda de um ou mais nodos de um sistema distribuído. Um tipo de implementação CFT é a duplicação de um componente em duas ou mais máquinas, de modo a garantir que algum problema em alguma delas não interrompa o funcionamento do sistema.

No entanto, uma classe de falhas geralmente não é considerada no projeto de muitos sistemas distribuídos: as falhas bizantinas, para as quais é necessária alguma implementação BFT (*byzantine-fault tolerance*). Esta classe de falhas é caracterizada pelo envio de informações conflitantes a diferentes componentes do sistema. Em seu artigo de 1982, Lamport *et al.* [24] descreve esta classe de falhas através de uma abstração com o exército bizantino, de onde vem o nome dessa classe de falhas. A situação é descrita como um determinado número de generais, estacionados com suas tropas ao redor de uma cidade a qual pretendem invadir. Se comunicando somente através de mensageiros, eles devem combinar sua estratégia de ataque. No entanto, alguns dos generais podem ser traidores, que tentarão confundir os outros, conforme exemplificado na Figura 2.1. Nessa situação, devem ser garantidas duas condições:

- Generais leais seguirão o mesmo plano de ação.
- Um pequeno número de traidores não pode levar os não-traidores a seguir uma má estratégia.

Os generais se comunicam através de mensagens orais, que são caracterizadas como:

1. Toda mensagem é entregue corretamente (não há perdas).
2. O receptor sabe quem enviou a mensagem.
3. A ausência de uma mensagem pode ser detectada.

No artigo é provado que para mensagens com essas características, não é possível atingir um consenso. O artigo então apresenta uma solução para o problema, onde os generais trocam informações entre pares e uma votação majoritária faz uma seleção entre todos os dados transmitidos. Essa solução requer que, para tolerar f generais traidores, são necessários $3f + 1$ generais ao todo.

No entanto, como demonstrado no artigo de Kevin Driscoll [14], o conhecimento acerca deste tipo de falha não é difundido na indústria, e é interpretada, geralmente, de forma equivocada, com probabilidade muito baixa de acontecer para demandar um tratamento específico, o qual geralmente é muito custoso.

Há uma tendência do aumento deste tipo de falha dada a evolução do hardware (que está cada vez menos confiável, menos durável, e demandando condições cada vez mais específicas para correto funcionamento, *e.g.* processadores com *clock* maior, litografia cada vez menor, sensíveis a variações de temperatura e alimentação elétrica) e da disseminação dos sistemas distribuídos.

O tratamento abstrato dos "generais bizantinos" prejudica a compreensão desse problema pela indústria, pois um engenheiro, que "sabe" que uma CPU não tem vontade própria e portanto não pode mentir, pode desconsiderar o conceito de "generais traidores" se escondendo em seus sistemas [14].

Além dessas ponderações, Driscoll [14] dá varios exemplos reais dessa classe de falhas. Um exemplo simples é a de um circuito digital que fica com um sinal "preso" em uma voltagem intermediária entre 0 e 1, que combinado com o ruído do circuito gera uma resposta diferente para cada leitura.

Uma técnica que fornece tolerância tanto a falhas *crash* quanto bizantinas é a Replicação Máquina de Estados (RME).

2.2 RME

Em uma implementação de Replicação Máquina de Estados [34], as réplicas devem apresentar a mesma evolução em seus estados: *(i)* partindo de um mesmo estado e *(ii)* executando o mesmo conjunto de requisições na mesma ordem, *(iii)* todas as réplicas devem chegar ao mesmo estado final, definindo o determinismo de réplicas. Para prover *(i)*, basta iniciar todas as réplicas com o mesmo estado (i.e., iniciar todas as variáveis que representam o estado com os mesmos valores nas diversas réplicas). Garantir o item *(ii)* envolve a utilização de um protocolo de difusão atômica. O problema da *difusão atômica* [21] (ou difusão com ordem total) consiste em fazer com que todos os processos corretos de um grupo entreguem todas as mensagens difundidas neste grupo

na mesma ordem. Já para prover *(iii)* é necessário que as operações executadas pelas réplicas sejam deterministas (i.e., a execução de uma mesma operação, com os mesmos parâmetros, deve produzir o mesmo resultado nas diversas réplicas).

2.2.1 Bft-SMaRt

O Bft-SMaRt [8] é uma biblioteca para implementação de aplicações através de Replicação Máquina de Estados [34] que tolera falhas bizantinas em algumas das réplicas (adicionalmente, o sistema pode ser configurado para tolerar apenas *crashes*). Esta biblioteca *open-source* de replicação foi desenvolvida em Java e implementa um protocolo para RME similar aos outros protocolos para tolerância a falhas bizantinas (ex.: [9]). Além disso, são fornecidos protocolos para reconfiguração e para gerenciamento de estados (*checkpoints*, atualização e transferência de estados).

O Bft-SMaRt assume um modelo de sistema usual para Replicação Máquina de Estados [8, 9]: $n \geq 3f + 1$ servidores para tolerar f falhas bizantinas; um número ilimitado de clientes que podem falhar; e um sistema parcialmente síncrono para garantir terminação. Estes parâmetros (n e f) podem ser alterados durante a execução através de reconfigurações [2]. Para comunicação, o sistema ainda necessita de canais ponto-a-ponto autenticados e confiáveis, que são implementados usando MACs (*message authentication codes*) sobre o TCP/IP. As chaves simétricas para a comunicação entre as réplicas são geradas através do protocolo *Signed Diffie-Helman* usando um par de chaves RSA para cada réplica. Já as chaves para a comunicação entre clientes e réplicas são geradas com base nos identificadores dos *endpoints* (cliente e réplica), i.e., não é necessário um par de chaves RSA para cada cliente. Além disso, pode-se configurar os clientes para assinar suas requisições, garantindo-se autenticação das requisições.

Resumidamente[1], cada réplica do Bft-SMaRt realiza as seguintes tarefas:

Recebimento de Requisições. Os clientes enviam suas requisições para as réplicas, que as armazenam em filas diferentes para cada cliente. A autenticidade das requisições é garantida por meio de assinaturas digitais, *i.e.*, os clientes assinam suas requisições. Desta forma, qualquer réplica é capaz de verificar a autenticidade das requisições e uma proposta para ordenação, a qual contém a requisição a ser ordenada, somente é aceita por uma réplica correta após a autenticidade desta requisição ser verificada.

[1]Uma descrição mais detalhada do Bft-SMaRt pode ser encontrada em [8].

Ordenamento de Requisições. Sempre que existirem requisições para serem executadas, uma instância do consenso é inicializada por uma réplica (chamada de líder) para definir uma ordem de entrega para um lote de requisições. Durante este processo, a réplica se comunica com as outras através de canais ponto-a-ponto confiáveis e autenticados. Caso uma requisição não seja ordenada dentro de um determinado tempo, o sistema troca a réplica líder. Um tempo limite para ordenação é associado a cada requisição r recebida em cada réplica i. Caso este tempo se esgotar, i envia r para todas as réplicas e define um novo tempo para sua ordenação. Isto garante que todas as réplicas recebem r, pois um cliente malicioso pode enviar r apenas para alguma(s) réplica(s), tentando forçar uma troca de líder. Caso este tempo se esgotar novamente, i solicita a troca de líder, que apenas é executada após $f + 1$ réplicas solicitarem esta troca, impedindo que uma réplica maliciosa force trocas de líder.

Execução de Requisições. Quando a ordem de execução de um lote de requisições é definida, este lote é adicionado em uma fila para então ser entregue à aplicação. Após o processamento da cada requisição, uma resposta é enviada ao cliente que solicitou tal requisição. O cliente, por sua vez, determina que uma resposta para sua requisição é válida assim que o mesmo receber pelo mesmo $f + 1$ respostas iguais, garantindo que pelo menos uma réplica correta obteve tal resposta.

Implementando Aplicações com o Bft-SMaRt

A forma de utilização do Bft-SMaRt, para programação de uma aplicação tolerante a falhas através de RME, é bastante simples. O Algoritmo 1 apresenta a API básica para clientes e servidores, mostrando a classe que deve ser instanciada pelos clientes para acessar o sistema, bem como a interface que deve ser estendida pelos servidores para implementar o serviço replicado.

Para acessar o serviço replicado, um cliente do Bft-SMaRt apenas deve instanciar uma classe `ServiceProxy` fornecendo seu identificador (inteiro) e um arquivo de configuração contendo o endereço (IP e porta) de cada um dos servidores. Após isso, sempre que o cliente desejar enviar alguma requisição para as réplicas (servidores), o mesmo deve invocar o método `invokeOrdered` especificando a requisição (serializada em um *array* de *bytes*). Para requisições que não precisam ordenação (ex.: operações de apenas leitura), o método `invokeUnordered` deve ser utilizado.

Por outro lado, para implementar o servidor, cada réplica deve estender a interface `Executable` e implementar o método abstrato `executeOrdered` que é invocado quando uma requisição deve ser executada. O método `executeUnordered` também deve ser

Algoritmo 1 API do BFT-SMART para clientes e servidores.

```
1  //API do Cliente
2  public class ServiceProxy {
3      public ServiceProxy(int id){
4          ...
5      }
6      public byte[] invokeOrdered(byte[] request){
7          ...
8      }
9      public byte[] invokeUnordered(byte[] request){
10         ...
11     }
12 }
13 //API do Servidor
14 public class MyServer extends Executable {
15     public MyServer(int id){
16         new ServiceReplica(id, this, ...) ;
17     }
18     public byte[] executeOrdered(byte[] request, MsgContext ctx){
19         //CÓDIGO DA APLICAÇÃO
20     }
21     public byte[] executeUnordered(byte[] request, MsgContext ctx){
22         //CÓDIGO DA APLICAÇÃO
23     }
24 }
```

implementado para execução de operações que não precisam ordenação. Além disso, é necessário instanciar uma `ServiceReplica` que representa propriamente a réplica, fornecendo o identificador (inteiro) da réplica que é mapeado para uma porta e endereço IP através de um arquivo de configuração.

Conforme já comentado, esta é a API básica do BFT-SMART que já é suficiente para implementar uma aplicação tolerante a falhas. No entanto, esta API é muito mais rica, apresentando também métodos para o gerenciamento do estado das réplicas (caso deseja-se empregar recuperação de réplicas). Uma descrição mais aprofundada sobre a API do BFT-SMART pode ser encontrada em [8].

2.3 Diversidade

A abordagem de usar diferentes implementações de um sistema (ou parte dele) para tolerar falhas de *software* foi proposta ainda na década de 70 [5, 33]. A ideia central destas propostas é que implementações diferentes não apresentem as mesmas falhas de *software*. Recentemente, vários trabalhos abordaram o uso de diversidade como medida

de segurança, pois apresenta vantagens como a *independência de falhas*, que garante que diferentes componentes ou réplicas não serão comprometidos simultaneamente, e com isto a *tolerância a intrusões*, que consiste na garantia de que o sistema continue operando corretamente mesmo que uma parte seja comprometida [7, 17, 18, 28, 30].

Um exemplo é a abordagem NVP (*N-version programming*)[11], que consiste na implementação de $N \geq 2$ versões funcionalmente equivalentes de um sistema e que seguem a mesma especificação. As diferentes versões são executadas concorrentemente e um algoritmo de votação é aplicado de modo que, na ocorrência de uma falha em uma das implementações, o resultado correto ainda assim possa ser utilizado e/ou uma resposta negativa seja fornecida, ao invés de um resultado potencialmente incorreto.

A justificativa é que à medida que um *software* aumenta em tamanho e complexidade se torna efetivamente impossível eliminar todos os defeitos antes de colocá-lo em uso. Além disso, a prática comum de duplicar instâncias do *software* para tolerar falhas ocasionais não resolve erros de *software*, que ocorrerão igualmente em todas as réplicas.

Existem vários pontos de um sistema (ou parte dele) que podem ser diversificados, formando os seguintes eixos de diversidade [28]:

1. Implementação: Consiste na implementação de diferentes versões de um *software*.
2. Administração: Consiste na distribuição dos componentes de um sistema em diferentes domínios administrativos.
3. Localização: Consiste em espalhar os vários componentes físicos de um sistema entre diferentes sítios de instalação.
4. COTS (*Commercial Off-The-Shelf*): Consiste em utilizar vários COTS diferentes que implementam algumas funcionalidades, como compiladores, bibliotecas, etc.
5. Sistema Operacional: Consiste na distribuição da aplicação entre diferentes sistemas operacionas.
6. Métodos: Consiste na utilização de métodos distintos para garantir algum atributo de segurança, como por exemplo cifrar sucessivamente um dado com mais de um algoritmo criptográfico.
7. *Hardware*: Consiste na distribuição da aplicação entre diferentes *hardwares*.

Porém, a diversificação de implementação ainda aumenta a segurança contra defeitos deliberados, inseridos intencionalmente, de forma maliciosa, pois provê as seguintes características [22]:

- A multiplicidade de implementações dificulta a inserção da mesma lógica maliciosa em todas as réplicas.
- Garante a completude, já que caso uma versão omita alguma ação, a consequente divergência dos estados é detectada.
- Componentes de *timeout* previnem que a ausência de resposta de um componente cause a parada do sistema (*i. e.*, um ataque de negação de serviço—DoS).

Um problema da diversidade de implementação é o seu alto custo de desenvolvimento, que é multiplicado por N [27]. Uma alternativa, no caso de *softwares* "de prateleira" como bancos de dados e sistemas de arquivos, é utilizar várias implementações de fornecedores diferentes e, através de uma camada de abstração, integrá-los com uma biblioteca BFT [10].

As soluções para diversidade têm diversas intersecções com as soluções BFT, principalmente:

- Execução em paralelo de réplicas que são funcionalmente equivalentes.
- Algoritmos de consenso para determinação da resposta correta.
- Necessidade de gerenciar a ordem da execução das tarefas para que todas as réplicas estejam sujeitas às mesmas transições de estado.

Portanto, é natural desejar a reutilização das implementações BFT para, também, implementar a diversidade e o NVP em um sistema distribuído.

Uma alternativa que oferece aumento considerável em segurança a baixo custo é a diversificação do sistema operacional [18, 27]. Isso é possível porque é muito baixo o número de vulnerabilidades que afetam simultaneamente mais de um sistema operacional. Num sistema sem essa diversidade, assim que um invasor compromete uma réplica, ele compromete todo o sistema, pois basta repetir o mesmo ataque para as demais.

2.4 Artefatos usados na implementação de diversidade no Bft-SMaRt

2.4.1 *Foreign Function Interface* (FFI)

Frequentemente é necessária a utilização de uma mistura de linguagens de programação: certa funcionalidade pode ser implementada em uma linguagem, enquanto utiliza

funções de um sistema legado ou uma biblioteca escrita em outra linguagem. Apesar de muitas vezes essa solução parecer "improvisada", é na realidade mais eficiente, pois possibilita focar no desenvolvimento de novas funcionalidades ao invés de reimplementar do zero *software* já existente [26].

É possível a utilização de mecanismos como *web services*, troca de mensagens ou *sockets* para fazer essa integração. Porém, além de manter dois recursos separados em execução, a comunicação entre os dois introduz um ponto de falha e de degradação de performance.

Para que as linguagens integradas sejam tratadas como um só programa, de tal modo que as duas linguagens de programação compartilhem recursos e memória diretamente, deve ser utilizado o mecanismo de *Foreign Function Interface* (FFI) ou "interface de função estrangeira". Uma FFI consiste de código que trata detalhes de baixo nível, por exemplo [6]:

- Encapsular objetos de e para código estrangeiro;
- Gerenciar alocação de memória; e
- Tratar diferentes convenções de chamada de função, argumentos implícitos, etc.

As FFIs se diferenciam de outros métodos de comunicação como IPC ou *sockets* pelo fato de que, durante a execução, o código das duas linguagens de programação farão parte de um mesmo processo, compartilhando o espaço de memória—as trocas de informação acontecem através de cópia de memória (*e.g. memcpy*), e portanto com pouco *overhead*.

Para cada linguagem, é necessária uma FFI correspondente. Linguagens de alto nível quase sempre vem acompanhadas de uma FFI para linguagem C, pois como os sistemas operacionais mais comuns possuem API em C, essa FFI é necessária para que a linguagem utilize os serviços do sistema operacional.

Java Native Interface (JNI)

A linguagem Java oferece duas maneiras principais de interação com código escrito em C: o JNI (Java Native Interface) [35] e o JNA (Java Native Access) [32].

O JNA é uma biblioteca de alto nível, cujo objetivo é permitir a um programa Java acessar bibliotecas escritas em C, porém não permite o acesso no sentido inverso, isto é, que um programa em C acesse uma biblioteca em Java [31], além de priorizar a facilidade de uso ao invés de performance [32].

O Java Native Interface (JNI) é uma interface de baixo nível, e é a fundação sobre a qual o JNA foi implementado. Permite fazer a interoperabilidade de programas escritos em linguagem Java com programas escritos em linguagem C de forma mais avançada, podendo carregar a máquina virtual e acessar métodos e objetos Java a partir de programas em C [35].

Essa interoperabilidade pode ser realizada de duas formas:

- O programa Java pode carregar, de modo dinâmico, uma biblioteca compartilhada (.so) escrita em C, e então executar suas funções.
- O programa C pode importar a `libjvm` e carregar uma JVM e então instanciar classes e invocar métodos Java - essa maneira de utilizar uma FFI é comumente chamada de *"embedding"*.

No entanto, a biblioteca C precisa ser escrita com suporte ao JNI, pois é necessário converter os dados de entrada (objetos) para uma forma com a qual o C possa trabalhar, e os resultados precisam ser formatados de forma adequada para que a JVM possa interpretá-los. Portanto, em geral, é desenvolvida uma "camada de tradução" para fazer a comunicação de um programa Java com bibliotecas C já existentes, ao invés de embutir tal tratamento nas próprias funções.

O JNI expõe diversos *typedefs* [29] correspondentes aos tipos de dado internos utilizados pela JVM, por exemplo:

- **jbyte** representa o tipo `byte` do java;
- **jobject** representa um objeto Java qualquer;
- **jbyteArray, jobjectArray, ...** representam um *array* do tipo especificado;
- ... entre outros.

Para a comunicação C → Java (C chamando métodos Java), o JNI fornece funções para trabalhar diretamente com os objetos Java. Por exemplo, para instanciar um objeto e chamar um de seus métodos é necessário [29]:

1. Buscar a classe do objeto, através da função `FindClass`.
2. Buscar o ID do construtor (i.e. um "ponteiro" para o construtor) através da função `GetMethodID`.
3. Converter os parâmetros para os tipos nativos Java. Por exemplo, se o construtor recebe como parâmetro uma *string*, é necessário construir um objeto `String`

(seguindo estes mesmos passos 1, 2, e 3), convertendo a codificação dos caracteres de ASCII (utilizado no C) para Unicode (utilizado no Java).

4. Executar o construtor utilizando a função `CallObjectMethod`.
5. Repetir o procedimento 2, desta vez para obter o método que se deseja executar.
6. Executar o método desejado, da mesma forma que o procedimento 4.
7. Converter o resultado da chamada para um tipo de dado que se possa trabalhar no C. Por exemplo se o método retornou *string*, necessita-se de realizar a conversão do item 3 em ordem reversa.

Para a comunicação Java → C (Java chamando o C), é necessário criar, no Java, as assinaturas dos métodos a serem chamados no C, com a palavra chave `native` [29]. Já em C, deve-se criar as funções que se deseja chamar correspondentes às criadas no Java, recebendo e retornando tipos específicos do Java, e com uma assinatura específica [35]. Para tanto existe o utilitário `javah` que gera arquivos .h com essas assinaturas. Neste trabalho, de modo a isentar o usuário de lidar com as estruturas do JNI, são utilizadas funções intermediárias entre a função do usuário e a função JNI.

Após o carregamento da JVM, essas funções devem ser registradas junto à JVM, fornecendo-lhe os ponteiros para as funções. Dentro de cada uma delas, assim como no método anterior, devem ser feitas a conversão de tipos entre o C e o Java.

Outra preocupação importante na utilização do JNI é o gerenciamento de alocação dos objetos. Trabalhando com o código nativo corre-se o risco de não liberar objetos que não estão mais em utilização e ocasionar um vazamento de memória [29].

`ctypes` - a FFI da linguagem Python

Assim como a JNI, a `ctypes` funciona através de ligação dinâmica com uma biblioteca compartilhada (`.so`). Após o carregamento da biblioteca, é necessário indicar ao `ctypes` as assinaturas das funções que se deseja utilizar.

Ao contrário da JNI, porém, não é necessário criar funções em C com assinaturas específicas para essa interação—o `ctypes` é capaz de utilizar diretamente as funções de C. Isto é possível pois o `ctypes` não permite que a linguagem C utilize serviços da linguagem Python, então o código em C não precisa ser modificado para dar suporte às estruturas da linguagem Python [16].

2.4.2 Protocol Buffers

Em um sistema diversificado um grande desafio é a troca de informações entre diferentes implementações que, apesar de atender aos mesmos requisitos possuem formas distintas de representação dos dados, que precisam ser normalizados para possibilitar o intercâmbio entre quaisquer duas implementações.

O Protocol Buffers é uma biblioteca desenvolvida pela Google em 2008 [37], cujo objetivo é serializar de forma universal dados em qualquer linguagem de programação. Isto é alcançado através da implementação de compiladores que, através de uma descrição das estruturas de dados, geram código específico para cada linguagem, possibilitando que o programador trabalhe utilizando as estruturas de dados nativas de sua linguagem e a biblioteca faz todo o trabalho para convertê-la em uma representação única. Além disso, a implementação é flexível e permite que novas linguagens sejam suportadas, através da implementação do respectivo compilador.

A forma como a informação a ser serializada é estruturada deve ser especificada através da definição de tipos de mensagens em arquivos `.proto`. Cada mensagem possui um ou mais campos unicamente numerados, e cada campo possui um nome e um tipo. É possível especificar campos como opcionais, obrigatórios, e repetidos (listas). Cada tipo de mensagem é um registro lógico contendo uma série de pares nome-valor, conforme o exemplo do Algoritmo 2 (adaptado de [19]).

Algoritmo 2 Definição de uma mensagem Protocol Buffers

```
1 message Pessoa {
2     required string nome = 1;
3     required int32 id = 2;
4     message Telefone {
5         required string number = 1;
6     }
7     repeated Telefone phone = 3;
8 }
```

No Algoritmo 2, é definida uma mensagem que transfere informações relativas a uma pessoa, que possui três atributos: um *id*, representado por um número inteiro, e um nome, representado por uma *string*. Ambos os campos são obrigatórios, sinalizado pela palavra-chave `required`. Além disso, uma lista de telefones é associada à pessoa. O Telefone é uma sub-entidade de Pessoa, e contém um único atributo, o número de

telefone, representado por uma *string*. A palavra chave `repeated` sinaliza que aquele atributo é uma lista. À direita de cada campo, são especificados números que os identificam unicamente, chamados *tags*. *Tags* de 1 a 15 precisam de um byte para serem codificados, *tags* de 16 a 2047 precisam de dois bytes, etc. Deste modo, campos mais utilizados devem utilizar *tags* menores, para economizar espaço ao transmitir a mensagem. Campos não-obrigatórios e não utilizados não são transmitidos.

Feitas as definições das mensagens, é executado o compilador que gera o código de acesso a dados na linguagem especificada, com métodos *get*/*set* para cada campo, além de métodos para ler e serializar toda a estrutura para um *array* de *bytes*.

Algoritmo 3 Exemplo de utilização de Protocol Buffers em C++

```
1 Pessoa fulano;
2 fulano.set_name("Jose da Silva");
3 fulano.set_id(1234);
4 fulano.SerializeToOstream(&output);
```

O código do Algoritmo 3, em C++, exemplifica a utilização da biblioteca para transmitir uma mensagem. Para cada entidade da mensagem é criada uma classe, com métodos para atribuir valores a cada campo da mesma. No caso, é criada uma instância da classe Pessoa, seu nome e *id* são configurados, e por fim a mensagem é serializada e enviada através do *stream* "output", que pode ser um arquivo, um soquete de rede, ou a saída do terminal.

Uma outra vantagem é que novos campos podem ser adicionados à mensagem sem quebra de compatibilidade – programas antigos simplesmente ignoram o novo campo quando é realizado o *parse* da mensagem.

Oficialmente são suportadas as linguagens C++ , Java e Python, e inúmeras outras foram implementadas pela comunidade *open-source* [19].

2.5 Considerações

Apesar dos potenciais ganhos em segurança e confiabilidade ao se utilizar uma arquitetura de Replicação Máquina de Estados, essa ainda é pouco utilizada na prática. O objetivo do Bft-SMaRt é popularizar esse tipo de aplicação, fornecendo uma biblioteca robusta e bem-documentada, abstraindo a complexidade envolvida nesse

tipo de protocolo [8]. Através da utilização de tecnologias estabelecidas como JNI e Protocol Buffers, este trabalho almeja a mesma facilidade para implementar aplicações diversificadas sobre o Bft-SMaRt.

Capítulo 3

Implementando Diversidade no Bft-SMaRt

Neste capítulo será descrito em detalhe como foi implementada a diversidade no Bft-SMaRt, incluindo arquitetura do código, seus módulos e como cada um foi implementado.

3.1 Princípios de projeto

Para implementar a diversidade no Bft-SMaRt, foram estabelecidas os seguintes princípios de desenvolvimento:

- A interface deve ser desenvolvida em linguagem C (já que a maioria das linguagens oferece alguma forma de interoperabilidade com C).
- As classes de dados (estados, requisições e respostas) devem ser serializadas para um protocolo comum, possibilitando a transferência de dados entre múltiplas linguagens.
- O programador usuário da solução não deve ser obrigado a programar em Java para utilizar a biblioteca.

Para a interoperabilidade entre o Bft-SMaRt e o C, será utilizado o JNI conforme descrito no capítulo anterior. Para atender à segunda premissa será utilizado o Protocol Buffers. Já a terceira premissa será alcançada através de uma arquitetura em camadas, onde a camada em Java fica transparente para o programador, através de camadas de tradução e abstração.

3.2 Arquitetura da solução proposta

Esta seção descreve a arquitetura proposta para implementação de diversidade em Replicação Máquina de Estados e discute como a mesma foi integrada no BFT-SMaRt, como possibilita às diferentes linguagens acessarem métodos e funções umas das outras, como são trocadas as informações (dados) entre elas, e as etapas para implementação de uma aplicação diversificada nesta arquitetura.

A solução proposta foi implementada em camadas, conforme a Figura 3.1.

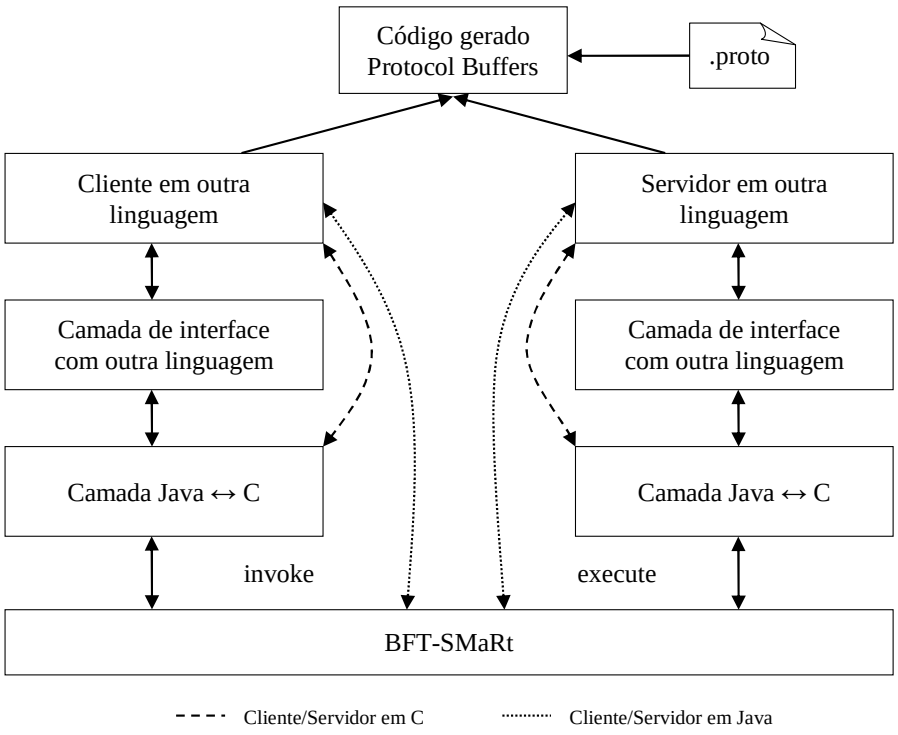

Figura 3.1: Camadas de interoperabilidade.

Para o envio de uma requisição do cliente ao servidor, cada camada é responsável pelas seguintes funcionalidades:

Cliente/servidor Essa é a parte escrita pelo usuário. Primeiramente as estruturas de dados são descritas na linguagem do Protocol Buffers (arquivo `.proto`), o qual é compilado e gera o código especializado de serialização e deserialização. Apenas *arrays* de *byte* são repassados entre as camadas.

Interface C ↔ outras linguagens Esta camada irá abstrair a conversão de estruturas de dados do C para as estruturas específicas das outras linguagens, e vice-versa. Essa camada permitirá que se implemente somente uma vez a conversão de tipos C/Java, que é a parte mais trabalhosa da comunicação, além de fornecer uma interface idiomática na linguagem de destino.

Interface Java ↔ C Realiza o tratamento das informações recebidas para que possam ser interpretados pela JVM. Esta camada recebe as informações da camada em C e comunica-se com o BFT-SMaRt.

No caso do recebimento das requisições pelo servidor e do recebimento da resposta pelo cliente, as camadas seguem a lógica inversa.

3.2.1 Camada Java ↔ C

A camada de interface Java ↔ C tem como objetivos o carregamento da JVM dentro do programa que a utiliza, a tradução e conversão das estruturas e tipos de dados nativos da linguagem C para as estruturas e tipos de dados Java e a chamada de funções e métodos, tanto no sentido C → Java (no caso do cliente), quanto no sentido Java → C (no caso do servidor).

As camadas em si foram implementadas utilizando funcionalidades do C++, por facilitar significativamente o trabalho com o JNI, e devem ser compiladas em um compilador apropriado. No entanto, a interface exposta por elas é utilizável a partir da linguagem C, pois foi empregada a cláusula `extern C`.

Interface de cliente

A interface de cliente possibilita a implementação de um cliente, em linguagem C, que irá utilizar um serviço implementado no BFT-SMaRt. Do lado Java foi criada a classe `ClientWrapper` (Algoritmo 4), que fornece uma interface simplificada para a classe `ServiceProxy` do BFT-SMaRt, de modo a facilitar a chamada a partir do C.

As operações e tipos de dados expostos pela interface de cliente (Algoritmo 5) são as seguintes:

BFT_BYTE corresponde ao tipo nativo `byte` em Java, que em C equivale a `signed char`.

setClasspath utilizada caso o usuário deseje utilizar um *classpath* customizado no momento de carregamento da JVM.

Algoritmo 4 Classe Java `ClientWrapper`

```
1  public class ClientWrapper {
2      private ServiceProxy proxy;
3      public ClientWrapper(int id) {
4          proxy = new ServiceProxy(id);
5      }
6      public byte[] executeOrdered(byte[] request) {
7          return this.proxy.invokeOrdered(request);
8      }
9      public byte[] executeUnordered(byte[] request) {
10         return this.proxy.invokeUnordered(request);
11     }
12 }
```

Algoritmo 5 Funções expostas para clientes em C

```
1 typedef signed char BFT_BYTE;
2 void setClasspath(const char* cp);
3 int createServiceProxy(int id);
4 int invokeOrdered
5       (BFT_BYTE command[], int tamanho, BFT_BYTE saida[]);
6 int invokeUnordered
7       (BFT_BYTE command[], int tamanho, BFT_BYTE saida[]);
8 void finalizeJVM();
9 int loadJVM();
```

invokeOrdered realiza uma chamada à função `executeOrdered` do BFT-SMaRt.

invokeUnordered realiza uma chamada à função `executeUnordered` do BFT-SMaRt.

finalizeJVM aguarda pelo término de todas as *threads* em execução na JVM e então a descarrega da memória.

loadJVM carrega a JVM no espaço de memória do programa.

createServiceProxy cria uma instância da classe `ClientWrapper`, que conecta aos servidores e fica aguardando chamadas.

As funções `invokeOrdered` e `invokeUnordered` no BFT-SMaRt recebem e retornam *arrays* de *byte*. Para implementar essa estrutura em C, foi seguida a convenção utilizada em funções como a `fread` e a `fwrite` – para a entrada de dados, a função recebe um ponteiro para o *array* de entrada e um inteiro com a quantidade de elementos a utilizar deste *array*. Para a saída de dados, a função recebe um ponteiro para o *array* de saída, que já deve estar alocado, e retorna um inteiro, que corresponde à quantidade de *bytes*

que foram escritos no *buffer*. Quando o usuário não desejar mais realizar chamadas, deve ser chamada a função `finalizeJVM`, para que a JVM possa ser finalizada de modo correto, com a finalização das conexões com as outras instâncias do BFT-SMaRt. Caso o usuário retorne da função `main` sem finalizar a JVM, a mesma será encerrada abruptamente, sem oportunidade de realizar mais procedimentos.

Interface de servidor

Para implementação de uma réplica servidor BFT, o BFT-SMaRt possibilita o desenvolvimento de uma implementação de uma classe abstrata, `DefaultSingleRecoverable`. Para uma comunicação Java ↔ C que possibilite a implementação de réplicas em outras linguagens, foi desenvolvida uma implementação mínima dos 4 métodos requeridos pela classe `DefaultSingleRecoverable`, através da classe `ServerWrapper` (Algoritmo 6), cada um apontando para um método "nativo" que, em tempo de execução, é associado a uma função em código C.

Algoritmo 6 Métodos nativos da classe `ServerWrapper`

```
1  public class ServerWrapper extends DefaultSingleRecoverable {
2      private native byte[] executeOrderedNative(byte[] commando);
3      private native byte[] executeUnorderedNative(byte[] commando);
4      private native byte[] getSnapshotNative();
5      private native void installSnapshotNative(byte[] commando);
6      @Override
7      public byte[] appExecuteOrdered(byte[] command,
8          MessageContext msgCtx) {
9          return executeOrderedNative(command);
10     }
11     @Override
12     public byte[] executeUnordered(byte[] command,
13         MessageContext msgCtx) {
14         return executeUnorderedNative(command);
15     }
16     @Override
17     public void installSnapshot(byte[] state) {
18         installSnapshotNative(state);
19     }
20     @Override
21     public byte[] getSnapshot() {
22         return getSnapshotNative();
23     }
24 }
```

Para a implementação da camada Java ↔ C de servidor, também foi adotado o C++ com a exposição da interface pública em C, como descrito anteriormente para o caso do cliente.

Algoritmo 7 Interface exposta para servidores em C

```
1 void implementExecuteOrdered
2         (int (*impl) (BFT_BYTE [], int, BFT_BYTE []));
3 void implementExecuteUnordered
4         (int (*impl) (BFT_BYTE [], int, BFT_BYTE []));
5 void implementInstallSnapshot(void (*impl) (BFT_BYTE [], int));
6 void implementgetSnapshot(int (*impl) (BFT_BYTE []));
7 int startServiceReplica(int id);
8 void finalizeJVM();
9 int loadJVM();
```

As operações e tipos de dados expostos pela interface de servidor (Algoritmo 7) são as seguintes:

implement* recebem como parâmetro um ponteiro para função que contém a implementação completa das respectivas operações. O usuário da biblioteca deve obrigatoriamente chamar essas funções antes de iniciar a réplica, pois em tempo de compilação a biblioteca de diversidade não possui referências para as funções que o usuário irá implementar futuramente. Portanto é necessário o "registro" das mesmas através de *callbacks*, prática comum em programas C [36]. Por exemplo, se a função que implementa as operações ordenadas em C se chamasse `execOrd`, então na inicialização deve ser executada a função `implementExecuteOrdered(&execOrd)`.

startServiceReplica inicia a réplica, conectando às outras instâncias e podendo receber conexões de clientes. A partir deste momento, as funções fornecidas previamente através de `implement*` serão invocadas pelo BFT-SMaRt.

3.2.2 Interface C ↔ outras linguagens

Para C e C++, a interface Java ↔ C pode ser utilizada diretamente, sem necessidade de camada intermediária, pois as linguagens são totalmente compatíveis. Para a utilização com outras linguagens de programação, deve ser criada uma camada adicional que utilize os serviços da camada Java ↔ C e exponha esse serviço para a linguagem

alvo. Neste trabalho foi criada uma camada para a linguagem Python, utilizando a interface `ctypes`. Cada linguagem precisará de uma implementação distinta, no entanto a forma como são utilizados os serviços da camada Java ↔ C será constante.

Como exemplo, discutiremos como foi desenvolvida a camada intermediária para Python. A utilização da interface Python é simples e o código se resumiu a implementação de duas classes:

BFTJVM classe abstrata, responsável por inicializar a JVM e configurar a assinatura das funções C (parâmetros e tipo de retorno). Uma listagem resumida desta classe é mostrada no Algoritmo 8.

BFTSMaRtServer classe abstrata, herdada da classe `BFTJVM`, que torna idiomática[1] a implementação das réplicas. Ao invés de lidar diretamente com as funções de C, o programador herda dessa classe implementando os métodos abstratos `executeOrdered` e `executeUnordered`, de forma análoga a uma implementação Java puro. Uma listagem resumida da classe `BFTSMaRtServer` é mostrada no Algoritmo 9.

Ao ser instanciada, a classe BFTJVM:

1. Carrega a biblioteca compartilhada *libbftsmr.so*, que consiste na camada Java ↔ C.
2. Configura o tipo de retorno das funções, através da propriedade `restype`. Não é necessário configurar os parâmetros das funções, que serão resolvidos dinamicamente a depender da maneira de como as funções forem chamadas.
3. Cria tipos que representam as funções de *callback* que serão chamadas pela camada em C, como por exemplo a `executeOrdered`, que será associada ao tipo `INVOKORDFUNC`, que retorna um inteiro (`c_int`) e recebe, nesta ordem, um ponteiro para char, um inteiro, e um ponteiro para ponteiro para char.

As funções `bftsmartallocate` e `bftsmartrelease`, definidas na camada em C, possibilitam ao Python alocar vetores em C (internamente utilizando malloc), já que Python não possibilita gerenciamento manual da memória.

A classe `BFTSMaRtServer` (Algoritmo 9) utiliza os tipos definidos em `BFTJVM` (Algoritmo 8) para encapsular funções Python em ponteiros para função C, estes em seguida são repassados ao código em C através da chamada `implementExecuteOrdered`.

[1] Um idioma é um meio padrão de se representar uma estrutura recorrente ou se resolver um problema em uma linguagem de programação específica, em geral utilizando funcionalidades específicas daquela linguagem [1]. Nesse contexto, uma implementação idiomática é aquela que utiliza plenamente as funcionalidades da linguagem de programação utilizada.

Algoritmo 8 Código da classe `BFTJVM` (simplificado–algumas funções omitidas)

```
1  class BFTJVM(object):
2      # parametros: dllpath: caminho para a biblioteca de diversidade
3      # classpth : classpath para inicializar a JVM.
4      def __init__(self, dllpath, classpth):
5              # carrega a biblioteca de diversidade
6              BFTJVM.libbft = CDLL(dllpath);
7              # configura o tipo de retorno de cada função em C.
8              # None equivale a void, e c_void_p equivale a (void *)
9              BFTJVM.libbft.setClasspath.restype = None
10             BFTJVM.libbft.bftsmartallocate.restype = c_void_p
11             BFTJVM.libbft.bftsmartrelease.restype = None
12             BFTJVM.libbft.finalizeJVM.restype = None
13             BFTJVM.libbft.implementExecuteOrdered.restype = None
14             # executa a função setClasspath, convertendo string Python para string C (char*)
15             BFTJVM.libbft.setClasspath(c_char_p(classpth));
16             # carrega a JVM
17             BFTJVM.libbft.loadJVM()
18 # define tipos de callback, que serão utilizados para criar ponteiros
19 # para função que permitem ao C executar as funções em Python
20 INVOKORDFUNC = CFUNCTYPE(c_int, POINTER(c_char), c_int,
       POINTER(POINTER(c_char)))
21 INVOKUNORDFUNC = CFUNCTYPE(c_int, POINTER(c_char), c_int,
       POINTER(POINTER(c_char)))
22 GETSNAPFUNC = CFUNCTYPE(c_int, POINTER(POINTER(c_char)))
```

Cada uma das funções que é chamada a partir do C realiza as seguintes operações:

1. Cria um *buffer* no espaço Python (`create_string_buffer`), e copia para ele o *array* de entrada (que vem do C).
2. Executa a função `invokeOrdered` do usuário.
3. Aloca um *array* de C, e copia o resultado do processamento da invokeOrdered (que vem em um *buffer* no espaço Python) para ele.
4. Retorna, por referência, o ponteiro para o *array* que contém o resultado.
5. Retorna o tamanho do *array* de resposta, possibilitando ao C percorrê-lo.

Por último, o construtor inicia o funcionamento da réplica, através da função `startServiceReplica`.

Algoritmo 9 Trecho da classe `BFTSMaRtServer`

```
1  # herda da classe BFTJVM
2  class BFTSMaRtServer(BFTJVM):
3  # parametro id: o id da replica como configurado no bft-smart.
4  def __init__(self,clspath,id,dllpath):
5          super(BFTSMaRtServer,self).__init__(dllpath,clspath)
6          # função que será chamada pelo C.Trata parametros
7          # de funcao, retorno e conversao de tipos, e chama a
8          # funcao da aplicacao, implementada pelo usuario.
9          def intermediateInvokeOrdered(input, size, outmem):
10             # recebe a requisição da camada C
11             p = create_string_buffer(size)
12             memmove(p, input, size)
13             result = self.invokeOrdered(p)  # chama a função do usuário
14             # converte e envia a resposta a camada C
15             output = BFTJVM.libbft.bftsmartallocate(len(result))
16             memmove(output, result, len(result))
17             outmem[0] = cast(output, POINTER(c_char));
18             return len(result)
19         # gera um ponteiro para função para intermediateInvokeOrdered.
20         self.invokeOrdCback = INVOKORDFUNC(intermediateInvokeOrdered)
21         # registra esse ponteiro para a camada C chamar posteriormente.
22         BFTJVM.libbft.implementExecuteOrdered(self.invokeOrdCback)
23         # inicia a réplica
24         BFTJVM.libbft.startServiceReplica(int(id))
```

3.2.3 Cliente/Servidor em outra linguagem

As etapas gerais para implementar uma aplicação diversificada utilizando a solução proposta neste trabalho são:

1. Projetar as mensagens que serão trocadas, através da criação dos arquivos `.proto`.
2. Compilar os arquivos `.proto`, gerando as classes de interface.
3. Programar a aplicação (cliente e servidor), nas linguagens escolhidas, utilizando as classes geradas no item 2.
4. Executar as réplicas e o cliente, passando os parâmetros apropriados.

A seguir descreve-se detalhamente a implementação de clientes e servidores em cada uma das linguagens implementadas neste trabalho. Um exemplo de implementação com trechos de código é apresentada na Seção 3.3.

Java

Protocol Buffers. Primeiramente é necessário compilar os arquivos `.proto` (Seção 2.4.2) do Protocol Buffers para gerar as classes que irão converter as mensagens em objetos Java e vice-versa. Para tal utiliza-se o comando:

```
protoc --java_out=diretorio_saida Mensagem.proto
```

As classes serão gravadas no diretório `diretorio_saida`, e podem ser compiladas como parte do projeto principal.

Servidor. Em sua forma mais simples, o servidor em Java é implementado através de herança da classe `DefaultSingleRecoverable`, que possui os métodos abstratos `executeOrdered`, `executeUnordered`, `getSnapshot` e `installSnapshot`, que devem ser implementados conforme a Seção 2.2.1 e [8].

Cliente. No programa cliente, deve ser instanciada a classe `ServiceProxy`, a qual fornece os métodos `invokeOrdered` e `invokeUnordered`. Esses métodos devem ser executados fornecendo o *array* de *bytes* proveniente do Protocol Buffers.

C

Protocol Buffers. Para gerar os arquivos .c e .h que fornecem as funções de conversão das mensagens em *structs* C e vice-versa. Para tal utiliza-se o comando:

```
protoc-c --c_out=diretorio_saida Mensagem.proto
```

Os arquivos gerados serão gravados no diretório `diretorio_saida`, e podem ser compilados como parte do projeto principal.

Servidor. Para implementar o servidor, é necessário o registro dos callbacks para as funções `executeOrdered` e `executeUnordered`. Após o registro, carrega-se a JVM através da função `loadJVM`, e executando a função `startServiceReplica` o servidor entra em operação.

Cliente. Para o cliente em C, é necessário carregar a JVM, após iniciar o proxy através da função `createServiceProxy`, e então pode-se utilizar as funções `invokeOrdered` e `invokeUnordered` para realizar as chamadas aos servidores.

C++

Protocol Buffers. Para gerar os arquivos .cc e .h que fornecem as funções de conversão das mensagens em objetos C++ e vice-versa. Para tal utiliza-se o comando:

```
protoc --cpp_out=diretorio_saida Mensagem.proto
```

Os arquivos gerados serão gravados no diretório `diretorio_saida`, e podem ser compilados como parte do projeto principal.

Servidor. Para implementar o servidor em C++, é utilizada a mesma biblioteca da implementação em C, portanto também é necessário o registro dos *callbacks* para as funções `executeOrdered` e `executeUnordered`. Após o registro, carrega-se a JVM através da função `loadJVM`, e executando a função `startServiceReplica` o servidor entra em operação. A diferença está na utilização do Protocol Buffers.

Cliente. Assim como para o cliente em C, é necessário carregar a JVM, após iniciar o proxy através da função `createServiceProxy`, e então pode-se utilizar as funções `invokeOrdered` e `invokeUnordered` para realizar as chamadas aos servidores.

Phyton

Protocol Buffers. Para gerar os arquivos .py que fornecem as funções de conversão das mensagens em objetos Python e vice-versa. Para tal utiliza-se o comando:

```
protoc --python_out=diretorio_saida Mensagem.proto
```

Os arquivos gerados serão gravados no diretório `diretorio_saida`, e podem ser compilados como parte do projeto principal.

Servidor. Para utilizar o servidor em Python, utiliza-se herança da classe `BFTSMaRt-Server`, e implementar os métodos abstratos assim como na implementação em Java. Ao instanciar essa classe, o servidor entra em execução, assim como na implementação em Java.

Cliente. Para utilizar o cliente em Python, instanciar a classe `BFTSMaRtClient`, e chamar seus métodos `invokeOrdered` e `invokeUnordered` de acordo com a necessidade.

3.3 Demonstração de funcionamento

Nessa seção será demonstrada a implementação e o funcionamento de uma aplicação com diversidade, utilizando as linguagens suportadas pela solução proposta neste trabalho (C, C++, Java e Python). Para tal, foi desenvolvida uma aplicação de lista com diversidade. Esta aplicação foi implementada nos servidores através de uma lista encadeada. Por brevidade, será mostrada a implementação de apenas uma das operações da lista (`ADD`).

3.3.1 Algoritmo da lista encadeada

A lista foi utilizada para armazenar inteiros e cinco operações foram implementadas para seu acesso: `ADD`, `REMOVE`, `GET`, `SIZE` e `CONTAINS`. A lista é representada por uma *struct* (ou classe, no caso de Java e Python) com dois ponteiros/referências: um para o início e outro para o fim da lista. A lista é simplesmente encadeada, com cada elemento contendo um ponteiro para o próximo.

A seguir, o funcionamento de cada operação será explicado detalhadamente.

CONTAINS

Realiza uma pesquisa linear na lista, percorrendo-a até o fim ou até encontrar o elemento procurado. Retorna 1 caso o inteiro passado como parâmetro exista na lista, e retorna 0 em caso contrário.

ADD

Inicialmente realiza uma pesquisa linear na lista, para verificar se o elemento a ser adicionado já não existe na lista. Caso não exista, o mesmo é incluído no fim da lista. Retorna 1 caso o elemento tenha sido incluído na lista, ou retorna 0 se o mesmo já existe.

REMOVE

Realiza uma pesquisa linear na lista, procurando o elemento a ser removido. Retorna 1 caso o elemento tenha sido encontrado e removido, ou retorna 0 caso o elemento não exista.

GET

Retorna o elemento da posição passada como parâmetro ou um código de erro caso não exista um elemento na posição indicada (posição solicitada maior que o tamanho da lista). Para isso, é realizada uma pesquisa linear com um contador, incrementado a cada elemento percorrido.

SIZE

Retorna o tamanho da lista. Para isto, a lista é percorrida com um contador, incrementado a cada elemento percorrido. Quando o fim da lista é atingido, o valor do contador é retornado.

3.3.2 Protocol Buffers

Foram criados três mensagens diferentes, `Estado` (Algoritmo 10), `Request` (Algoritmo 11) e `Response` (Algoritmo 12), representado o Estado da réplica, as requisições e respostas, respectivamente. A palavra-chave `package` define um espaço de nome a ser usado para a classe criada (e.g. em Java, um *package* e em C++, um *namespace*).

Após criados os arquivos `.proto` do Protocol Buffers, os mesmos devem ser compilados para cada uma das linguagens alvo, conforme discutido na Seção 3.2.3. Os arquivos de código-fonte gerados são então utilizados na implementação dos clientes e servidores.

Algoritmo 10 Definição Protocol Buffers para a mensagem `Estado`

```
1 package bftbench;
2 message Estado
3 {
4         repeated int32  lista  = 1;
5 }
```

3.3.3 Cliente

Os clientes foram implementados para funcionar de forma interativa, *i.e.* solicitando uma entrada do usuário e apresentando o resultado na tela, com exceção do cliente Java, que foi desenvolvido para realizar os *benchmarks* do Capítulo 4.

Algoritmo 11 Definição Protocol Buffers para a mensagem `Request`

```
1  package bftbench;
2  message Request
3  {
4          enum RequestType {
5                  ADD = 0;
6                  REMOVE = 1;
7                  SIZE = 2;
8                  CONTAINS = 3;
9                  GET = 4;
10         }
11         required RequestType action = 1;
12         optional int32 value = 2;
13 }
```

Algoritmo 12 Definição Protocol Buffers para a mensagem `Response`

```
1  package bftbench;
2  message Response
3  {
4          optional bool BoolResponse = 1;
5          optional int32 IntResponse = 2;
6  }
```

O cliente em Java ficou responsável pela coleta de dados dos *benchmarks*, e foi baseado em clientes já existentes como parte do projeto Bft-SMaRt. Mais detalhes sobre seu funcionamento estão disponíveis no Capítulo 4.

De modo a testar o funcionamento da interface com as linguagens C e C++, foi criado um cliente em C. Esse cliente está dividido em duas partes: inicialização (Algoritmo 13), onde é configurado e carregado o Bft-SMaRt, e execução (Algoritmo 14), onde são feitas as requisições aos servidores. Não foi necessário construir um cliente em C++, pois a interface utilizada em ambos é a mesma.

Por fim foi criado um cliente com a linguagem Python (Algoritmo 15), com a mesma funcionalidade que o cliente em C, solicitando entrada do usuário para executar as operações na lista interativamente.

3.3.4 Servidor

Na implementação de lista nos servidores, houve o cuidado de utilizar o mesmo algoritmo de lista em todas as linguagens (Algoritmos 16 e 17), de modo a garantir que a complexidade das operações fosse a mesma e não distorcesse os valores aferidos

Algoritmo 13 Código de um cliente em C (inicialização do BFT-SMART, somente operação `ADD`)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <bftsmart-wrapper.h>
4  #include <Request.pb-c.h>
5  #include <Response.pb-c.h>
6  #include <Estado.pb-c.h>
7  int main(int argc, char* argv[]) {
8  // argv[1]: id da réplica, argv[2]: classpath para o Java
9      setClasspath(argv[2]);
10     loadJVM();
11     if (createServiceProxy(atoi(argv[1])) != 0) {
12         printf("%s","Erro ao criar service proxy.\n"); return -2;
13     }
14     BFT_BYTE saida[10000]; // o retorno da chamada é escrito aqui
15     int cmd,result;
16     // lista as operações disponíveis para o usuário
17     do {
18         printf("%s\n", "1-add");
19         printf("%s\n", "2-get");
20         printf("%s\n", "3-remove");
21         printf("%s\n", "4-contains");
22         printf("%s\n", "5-size");
23         printf("%s\n", "6-FINALIZAR. pressione ctrl-c depois.");
```

nos experimentos. Desta forma, apesar de em linguagens como Java, C++ e Python existirem estruturas específicas da linguagem de programação que realizem todas as operações, foi implementada a lista encadeada utilizando *structs* e ponteiros em C e C++, e classes e referências em Python e Java.

O servidor em C é dividido em três partes: inicialização (Algoritmo 18), onde é carregado o BFT-SMART, execução (Algoritmo 19), onde são processadas as requisições, e transferência de estado (Algoritmo 20), onde são processadas a criação e instalação de estado enviado pelo BFT-SMART durante processo de verificação e recuperação de réplicas.

A implementação do servidor em Python também possui as mesmas partes: inicialização (Algoritmo 21), execução e transferência de estado (Algoritmo 22), porém bem mais enxutas, devido às funcionalidades da linguagem (dinâmica e orientada a objeto) e parte da inicialização ter sido integrada à classe `BFTJVM` (Algoritmo 8).

Algoritmo 14 Código de um cliente em C (requisição ao servidor, somente operação `ADD`)

```
1       // continua do Algoritmo 13
2       // lê a opção do usuário
3       scanf("%d",&cmd);
4       getchar();
5       // variáveis para as mensagens do Protocol Buffers
6       Bftbench__Request req ; Bftbench__Response *rsp ;
7       unsigned int tamanho ;
8       BFT_BYTE * out ;
9       switch (cmd) {
10      case 1:
11              printf("%s\n", "Digite o valor:"); scanf("%d", &cmd); getchar();
12              // cria a mensagem de requisição
13              bftbench__request__init (&req);
14              req.value = cmd;
15              req.has_value= 1;
16              req.action = BFTBENCH__REQUEST__REQUEST_TYPE__ADD;
17              // converte a mensagem para array de bytes.
18              tamanho = bftbench__request__get_packed_size(&req);
19              out = (BFT_BYTE*) malloc (tamanho);
20              bftbench__request__pack(&req, (uint8_t*) out);
21              // executa a operação no servidor
22              result = invokeOrdered(out, tamanho, saida);
23              // le a resposta da operacao e imprime na tela
24              rsp = bftbench__response__unpack(NULL, result, (const uint8_t*)saida);
25              printf("%s%d\n","response:",rsp->boolresponse);
26              // libera a memória alocada
27              bftbench__response__free_unpacked(rsp, NULL);
28              free(out);
29              break;
30      case 2: // ...
31      case 3: // ...
32      case 4: // ...
33      case 5: // ...
34      }
35  } while (cmd != 6);
36  finalizeJVM();
37  return 0;
38 }
```

3.3.5 Execução

Para a execução da aplicação, é necessário primeiro iniciar as 4 réplicas (servidores). Após iniciadas as réplicas, são iniciados clientes e realizados testes para verificar se a execução é bem sucedida e se os valores retornados pelas operações são coerentes. Foram realizados testes tanto em ambientes sem diversidade, com todos os servidores

Algoritmo 15 Código de um cliente em Python, somente operação `ADD`

```
1  #!/usr/bin/python
2  from ctypes import *
3  from bftsmart_clientlib import *
4  import Estado_pb2
5  import Request_pb2
6  import Response_pb2
7  # codigo main do programa
8  bc = BFTSMaRtClient(sys.argv[1], sys.argv[2], sys.argv[3])
9  # argv[1]: classpath para o java. argv[2]: id da réplica. argv[3]: caminho da biblioteca de diversidade
10 # apresenta as opções para o usuário
11 mydata = raw_input('acao(add,remove,size,contains,get,sair) :')
12 while mydata != 'sair':
13     if mydata == 'add':
14         # cria a mensagem, preenche com um numero digitado pelo usuário
15         req = Request_pb2.Request()
16         req.action = Request_pb2.Request.ADD
17         req.value = int(raw_input('numero:'))
18         # executa operação no servidor
19         rsp = bc.invokeOrdered(req.SerializeToString())
20         # lê a resposta e imprime na tela
21         rspP = Response_pb2.Response()
22         rspP.ParseFromString(rsp)
23         print 'resposta:'
24         print rspP.BoolResponse
25     if mydata == 'remove':
26         # ...
27     if mydata == 'size':
28         # ...
29     if mydata == 'contains':
30         # ...
31     if mydata == 'get':
32         # ...
33     mydata = raw_input('acao(add,remove,size,contains,get,sair) :')
34 print 'pressione ctrl-c.'
35 bc.finalizeJVM()
```

na mesma linguagem, e em ambiente com diversidade, com cada servidor em uma linguagem diferente. Para validar o funcionamento dos protocolos de transferência de estado, cada uma das réplicas era reiniciada, alternadamente, após a realização de um conjunto de operações na lista.

Na invocação da biblioteca desenvolvida, é papel do usuário fornecer o caminho para a *classpath* onde as classes do BFT-SMaRt estarão armazenadas. Além disso, na biblioteca Python, é necessário fornecer o caminho para a biblioteca compartilhada *libbftsmr.so*, já que por ser uma linguagem interpretada, o carregamento da biblioteca

Algoritmo 16 Implementação de lista em C, somente operação `ADD`

```
1  typedef struct elemento {
2      int dado;
3      struct elemento * proximo;
4  } t_elemento;
5  typedef struct lista {
6      t_elemento * inicio;
7      t_elemento * fim;
8  } t_lista;
9  int insereFinal(int valor, t_lista * l) {
10     t_elemento * novoultimo = (t_elemento *)malloc(sizeof(t_elemento));
11     novoultimo->dado = valor;
12     novoultimo->proximo = NULL;
13     if(l->inicio == NULL)
14         l->inicio = novoultimo;
15     else
16         l->fim->proximo = novoultimo;
17     l->fim = novoultimo;
18     return 0;
19 }
20 int buscarIndice(int valor, t_lista * l) {
21     t_elemento * atual = l->inicio;
22     int i = 0;
23     while (atual != NULL) {
24         if (atual->dado == valor) return i;
25         i++;
26         atual = atual->proximo;
27     }
28     return -1;
29 }
```

é feito em tempo de execução.

3.4 Considerações

Com a implementação da aplicação de lista, apesar de simples, foi possível validar que a implementação funciona corretamente. Como o sistema continuou operando normalmente e retornando resultados corretos, ficou evidenciado que o estado era transferido corretamente entre as réplicas.

Algoritmo 17 Implementação da lista em Python, somente operação `ADD`

```
1  class Elemento(object):
2      valor = 0
3      proximo = None;
4  class ListaNormal(object):
5      inicio = None
6      fim = None
7      def insereFinal(self, valor):
8          novoultimo = Elemento()
9          novoultimo.valor = valor
10         novoultimo.proximo = None
11         if self.inicio is None:
12             self.inicio = novoultimo
13         else:
14             self.fim.proximo = novoultimo
15         self.fim = novoultimo
16         return 0
17     def buscarIndice(self, valor):
18         atual = self.inicio
19         i = 0
20         while atual is not None:
21             if atual.valor == valor:
22                 return i
23             i += 1
24             atual = atual.proximo
25         return -1
```

Algoritmo 18 Servidor em C (inicialização)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <bftsmart-wrapper.h>
4  #include <Request.pb-c.h>
5  #include <Response.pb-c.h>
6  #include <Estado.pb-c.h>
7  // omitida a implementação de funções do servidor (Algoritmos 19 e 20).
8  int main(int argc, char* argv[]) {
9      // argv[1]: id da replica, argv[2]: classpath para o Java
10     // cria a estrutura de lista, usada no Algoritmo 19
11     state = criarLista();
12     // inicia a JVM
13     setClasspath(argv[2]);
14     loadJVM();
15     // registra as funções do servidor junto ao bft-smart
16     implementExecuteOrdered(&execute);
17     implementExecuteUnordered(&execute);
18     implementInstallSnapshot(&installSnap);
19     implementgetSnapshot(&getSnap);
20     // inicia a réplica
21     startServiceReplica(atoi(argv[1]));
22     finalizeJVM();
23     return 0;
24 }
```

Algoritmo 19 Servidor em C, somente operação `ADD` da lista.

```
1  // .... omitida a implementação da lista (Algoritmo 16)
2  t_lista * state;
3  int execute(BFT_BYTE cmd[], int siz, BFT_BYTE ** mem) {
4      // lê a mensagem de requisição, convertendo de array de bytes para Protocol Buffers, determina
           a ação a ser executada, e a executa.
5      Bftbench__Request *msg = bftbench__request__unpack(NULL, siz, (const uint8_t*)cmd);
6      Bftbench__Response rsp = BFTBENCH__RESPONSE__INIT;
7      int x;
8      switch (msg->action)
9      {
10         // executa a inserção na lista e retorna se foi bem-sucedido
11         // trabalha diretamente com os objetos Protocol Buffers
12         case BFTBENCH__REQUEST__REQUEST_TYPE__ADD:
13             x = buscarIndice(msg->value, state);
14             rsp.has_boolresponse = 1;
15             rsp.boolresponse = 0;
16             if (x < 0) {
17               rsp.boolresponse = 1;
18               insereFinal(msg->value, state);
19             }
20             break;
21         // case REMOVE, case CONTAINS, case SIZE e case GET omitidos.
22     }
23     // converte a resposta em array de bytes, e retorna para a função chamadora
24     unsigned int tamanho = bftbench__response__get_packed_size(&rsp);
25     BFT_BYTE * out = (BFT_BYTE*) malloc (tamanho);
26     bftbench__response__pack(&rsp, (uint8_t*) out);
27     (*mem) = out;
28     return tamanho;
29 }
```

Algoritmo 20 Servidor em C (funções de transferência de estado)

```
1  void installSnap(BFT_BYTE stateNovo[], int siz) {
2      // abre a mensagem Protocol Buffers
3      Bftbench__Estado *msg = bftbench__estado__unpack(NULL, siz, (const uint8_t*)stateNovo);
4      // destrói e recria a lista com os valores recebidos
5      destruirLista(state);
6      state = criarLista();
7      unsigned int i;
8      for (i = 0; i < msg->n_lista; i++) {
9          insereFinal(msg->lista[i], state);
10     }
11     // libera a memória alocada
12     bftbench__estado__free_unpacked(msg, NULL);
13 }
14 int getSnap(BFT_BYTE ** mem) {
15     // cria uma mensagem vazia Protocol Buffers e aloca espaço para a lista
16     Bftbench__Estado est = BFTBENCH__ESTADO__INIT;
17     int x = tamanhoLista(state);
18     est.n_lista = x;
19     est.lista = (int32_t*) malloc(sizeof(int) * est.n_lista);
20     // copia todos os elementos da lista para a mensagem Protocol Buffers
21     int i = 0;
22     t_elemento * atual = state->inicio;
23     while (atual != NULL) {
24         est.lista[i] = atual->dado;
25         i++;
26         atual = atual->proximo;
27     }
28     // converte a mensagem para array de bytes e retorna ao bft-smart.
29     unsigned int tamanho = bftbench__estado__get_packed_size(&est);
30     BFT_BYTE * out = (BFT_BYTE*) malloc (tamanho);
31     bftbench__estado__pack(&est, (uint8_t*)out);
32     free(est.lista);
33     (*mem) = out;
34     return tamanho;
35 }
```

Algoritmo 21 *Script* principal (*main*) do servidor em Python.

```
1 import sys
2 from list_imp import *
3 # argv[1]: classpath do Java. argv[2]: id da réplica. argv[3]: caminho da biblioteca de diversidade.
4 # inicia o servidor.
5 bc = BFTList(sys.argv[1], sys.argv[2], sys.argv[3])
6 bc.finalizeJVM()
```

Algoritmo 22 Implementação do servidor em Python, somente operação `ADD`

```
1  import sys
2  import Estado_pb2
3  import Request_pb2
4  import Response_pb2
5  from bftsmart_serverlib import *
6  class BFTList(BFTSMaRtServer):
7      def __init__(self,clspath, id, dllpath):
8          # chama o construtor da classe base
9          super(BFTList,self).__init__(clspath, id, dllpath)
10         self.state = ListaNormal()
11     def execute(self, input):
12         req = Request_pb2.Request()
13         req.ParseFromString(input)
14         if req.action == Request_pb2.Request.ADD:
15             val = req.value;
16             x = self.state.buscarIndice(val)
17             rsp = Response_pb2.Response()
18             rsp.BoolResponse = False
19             if x < 0:
20               rsp.BoolResponse = True
21               self.state.insereFinal(val)
22             return rsp.SerializeToString()
23     def invokeOrdered(self, input):
24         return self.execute(input)
25     def invokeUnordered(self, input):
26         return self.execute(input)
27     # lê as informações da lista e retorna para o bft-smart.
28     def getSnapshot(self):
29         est = Estado_pb2.Estado()
30         atual = self.state.inicio
31         while atual is not None:
32             est.lista.append(atual.valor)
33             atual = atual.proximo
34         return est.SerializeToString()
35     # lê as informações recebidas e re-cria a lista com elas.
36     def installSnapshot(self, input):
37         est = Estado_pb2.Estado()
38         est.ParseFromString(input)
39         self.state = ListaNormal()
40         for i in est.lista:
41             self.state.insereFinal(i)
```

Capítulo 4

Experimentos

Visando analisar o desempenho da arquitetura proposta e da implementação desenvolvida, bem como o comportamento de uma Replicação Máquina de Estados com diversidade, alguns experimentos foram realizados no Emulab [39]. Neste capítulo são descritos os experimentos realizados, seu ambiente de execução e os resultados coletados.

4.1 Aplicação

Foram desenvolvidos dois testes: um teste nulo (requisição e resposta vazios) para avaliar o *overhead* da implementação, e uma aplicação de lista foi desenvolvida para ter seu desempenho analisado—suas implementações estão descritas no Capítulo 3.

Cada servidor executou em uma máquina separada, enquanto que 100 clientes executaram na outra máquina. Todos os experimentos realizados tiveram uma fase de *warm-up* que consiste na inicialização da lista com 200k operações `ADD`—com exceção do *warm-up* da própria operação de `ADD`, que consistiu na execução aleatória de operações dos outros tipos, e nesse caso o *warm-up* executa 10% do número total de operações `ADD` executadas.

Em nossos experimentos, a lista foi inicializada com 200k entradas em cada réplica (2k entradas por cliente) e os valores para as operações `ADD`, `GET` e `REMOVE` foram selecionados aleatoriamente seguindo uma distribuição uniforme, com valores de 0 a 400k. Além disso, todos os clientes executados foram desenvolvidos na linguagem Java, enquanto que os servidores foram diversificados conforme os cenários descritos nas seções seguintes.

A latência e o *throughput* destas operações foram determinados, com o objetivo de comparar o desempenho do sistema nos diversos cenários. A latência foi medida em

um dos clientes e os valores apresentados representam a média de 1000 execuções, excluindo-se 10% dos valores com maior desvio. Já o *throughput* apresentado é o pico atingido pelos servidores, medido no servidor líder do consenso [8] a cada 5000 requisições.

4.2 Análise do Desempenho: Sem Diversidade no Ambiente de Execução

Primeiramente, executamos alguns experimentos em um ambiente onde todas as máquinas possuiam a mesma configuração, o que possibilitou uma análise mais precisa a respeito do *overhead* introduzido pelo uso de diversidade. Desta forma, o ambiente foi consistindo por 5 máquinas *d710* (2.4 GHz 64-*bit* Intel Quad Core Xeon E5530, 12GB de RAM e interface de rede gigabit) conectadas a um *switch* de 1Gbps. O ambiente de *software* utilizado foi o sistema operacional Ubuntu 12.04 com *kernel* 3.2.0, JVM Oracle JDK 1.7.0_79 (Java), g++ 4.6.0 (C++), gcc 4.8.3 (C) e Python 2.7.3 (Python). O Bft-SMaRt foi configurado com $n = 4$ servidores para tolerar até uma falha Bizantina.

4.2.1 Resultados e Análises

A Tabela 4.1 apresenta os valores para o *throughput* e a latência em três cenários distintos: *(1)* apenas usando Java, i.e., o Bft-SMaRt sem diversidade; *(2)* usando Java e Protocol Buffers (Java + PB), que possibilita a avaliação do *overhead* introduzido pelo mecanismo de representação dos dados; e *(3)* usando cada réplica do Bft-SMaRt em uma linguagem diferente (Java, C, C++ e Python), que possibilita avaliar toda a arquitetura proposta, incluindo o *overhead* tanto do mecanismo de representação dos dados quanto para uma linguagem executar métodos ou funções de outras linguagens.

Podemos perceber que a variação no desempenho introduzida pela utilização de Protocol Buffers é praticamente desprezível. Porém, o *throughput* diminui quando o sistema é configurado com uma réplica em cada linguagem, devido às variações de performance específicas a cada uma delas. Este comportamento fica mais evidente para as operações `GET` e `REMOVE`. A latência também se comportou de forma semelhante, aumentando quando réplicas em diferentes linguagens foram utilizadas.

Visando analisar os fatores que levaram a esta queda de desempenho, medimos o tempo que cada operação leva para ser executada em cada implementação (execução da operação `executeOrdered` na linguagem específica para as operação `ADD`, `GET` e `REMOVE`).

		Java	Java + PB	Java, C, C++ e Python	Java, C, C++ e C++
ADD	Throughput (kop/seg)	11.55	11.32	9.25	10.59
	Latência (ms)	34.92	35.56	100.43	47.46
	Desvio Padrão (ms)	10.10	9.22	46.15	15.48
GET	Throughput (kop/seg)	4.45	4.40	0.85	2.08
	Latência (ms)	24.33	24.08	126.63	55.61
	Desvio Padrão (ms)	2.06	2.22	7.73	5.97
REMOVE	Throughput (kop/seg)	2.66	2.55	0.48	1.22
	Latência (ms)	47.06	47.91	231.23	90.44
	Desvio Padrão (ms)	5.01	4.19	9.99	7.19

Tabela 4.1: Experimentos sem diversidade no ambiente de execução.

Como podemos observar na Tabela 4.2, a implementação em Python apresentou um tempo de resposta muito superior as demais, enquanto C e C++ tiveram desempenhos próximos mas que são praticamente o dobro do que em Java. Vale destacar que estes valores não sofrem influência da arquitetura para diversidade proposta neste trabalho, pois estes valores foram medidos já na linguagem específica (Java, C, C++ e Python).

Como a réplica em Python possui performance mais baixa, ficando muito atrasada em relação às demais, o BFT-SMaRt iniciava os protocolos para transferência e atualização de estados [8]. Durante este procedimento, esta réplica solicitava o estado que era transferido a partir das outras réplicas, o que acaba impactando no desempenho do sistema. Para evidenciar este comportamento, na última coluna da Tabela 4.1 são apresentados os resultados para estes experimentos trocando a réplica em Python por outra réplica em C++. Comparando com a utilização de apenas Java, podemos perceber que neste caso o desempenho melhora, ficando muito próximo para a operação de `ADD` e praticamente a metade para as outras operações, o que é explicado pelo fato de as operações demorarem praticamente o dobro em C++ (Tabela 4.2). O *throughput* da operação `ADD` fica muito próximo da configuração com apenas Java porque os resultados apresentados referem-se ao pico atingido pelo sistema e, como inicialmente a lista está vazia, este valor é conseguido logo no começo (a queda no desempenho em Python está relacionada com as buscas que ocorrem na lista).

Para certificar-se de que a queda de desempenho está relacionada com o processamento da requisição em uma determinada linguagem, analisamos uma aplicação vazia (nada é processado nos servidores, que apenas retornam uma resposta). Este tipo de aplicação é comumente utilizado para avaliar estes sistemas [8] e apenas os tamanhos das requisições/respostas são configurados. A Tabela 4.3 apresenta os resultados para a configuração com requisições/respostas de tamanho 0/0. Podemos perceber que neste

		Java	C	C++	Python
`ADD`	Tempo de execução (ms)	0.33	0.71	0.55	11.45
	Desvio Padrão (ms)	0.09	0.17	0.13	2.45
`GET`	Tempo de execução (ms)	0.20	0.70	0.43	7.68
	Desvio Padrão (ms)	0.10	0.32	0.10	3.36
`REMOVE`	Tempo de execução (ms)	0.51	1.09	0.67	14.03
	Desvio Padrão (ms)	0.06	0.19	0.12	0.51

Tabela 4.2: Tempo de resposta das linguagens (execução de *executeOrdered*).

caso o desempenho é praticamente o mesmo em todos os cenários, evidenciando que o *overhead* da implementação é negligível.

	Java	Java + PB	Java, C, C++ e Python
Throughput (kops/seg)	48.55	48.35	47.17
Latência (ms)	2.24	2.28	2.29
Desvio Padrão (ms)	0.23	0.29	0.36

Tabela 4.3: Experimento para um aplicação vazia (0/0).

4.3 Análise da Segurança: Com Diversidade no Ambiente de Execução

Apesar da aplicação apresentar diversidade de implementação nos experimentos anteriores, a mesma configuração de execução foi utilizada em cada servidor de forma que uma mesma vulnerabilidade pode comprometer toda a aplicação. Por exemplo, uma vulnerabilidade no sistema operacional utilizado (Ubuntu) pode ser explorada em todas as réplicas, visto que todas executam sobre este mesmo sistema operacional. Neste sentido, esta seção apresenta experimentos executados em um ambiente que contempla outros eixos de diversidade, o que aumenta a segurança das aplicações conforme discutido a seguir.

4.3.1 Configuração do Ambiente de Execução e Análise da Segurança.

Visando aumentar a segurança através de diversidade, os ambientes de execução das réplicas devem ser diferentes, de forma que uma mesma vulnerabilidade não esteja presente em mais de um deles. A Tabela 4.4 apresenta a configuração adotada em cada réplica, que novamente foram conectadas a um *switch* de 1Gbps. A nomenclatura utilizada para o *hardware* é aquela fornecida pelo Emulab [39]. Dentre as configurações disponibilizadas pela plataforma, utilizamos uma configuração diferente para cada servidor. Também utilizamos diferentes distribuições e/ou versões de sistemas operacionais, além de compiladores e/ou ambientes de execução para as linguagens utilizadas. O C++ está presente em todas as réplicas (menos em Java) visto que a camada intermediária foi desenvolvida em C++ e teve sua interface exportada para C (Capítulo 3).

Eixo de Diversidade		Réplica 1	Réplica 2	Réplica 3	Réplica 4
Aplicação		Java	C++	C	Python
Sistema Operacional		Fedora Core 15 *kernel* 2.6.40	FreeBSD 10.0	CentOS 7.1 *kernel* 3.10.0	Ubuntu 12.04 *kernel* 3.2.0
COTS	JVM (Java)	Oracle JDK 1.7.0_79	OpenJDK 1.8.0_60	Oracle JDK 1.8.0_60	OpenJDK 1.7.0_91
	C	–	–	gcc 4.8.3	–
	C++	–	clang 3.3	g++ 4.8.3	g++ 4.6.3
	Python	–	–	–	Python 2.7.3
Hardware		pc2400w	pc3000	d820	d710

Tabela 4.4: Configuração das réplicas com diversidade.

Como podemos perceber, um atacante não é capaz de explorar uma mesma vulnerabilidade para comprometer mais de uma réplica. Por exemplo, uma vulnerabilidade no sistema operacional Ubuntu comprometeria apenas a réplica 4, enquanto que uma falha no *hardware* pc3000 afetaria apenas a réplica 2, e assim por diante. Desta forma, a segurança da aplicação é aumentada na medida em que mais vulnerabilidades são necessárias para comprometer o sistema. De fato, é necessário o comprometimento de mais de uma réplica para que o sistema deixe de funcionar corretamente.

		Java	Java + PB	Java, C, C++ e Python
ADD	Throughput (kop/seg)	4.97	4.62	1.16
	Latência (ms)	27.32	29.69	131.74
	Desvio Padrão (ms)	6.52	8.89	41.70
GET	Throughput (kop/seg)	10.75	10.82	1.54
	Latência (ms)	9.75	9.55	71.02
	Desvio Padrão (ms)	1.10	0.97	8.40
REMOVE	Throughput (kop/seg)	6.61	6.29	1.05
	Latência (ms)	16.18	16.72	106.63
	Desvio Padrão (ms)	1.68	1.176	13.28

Tabela 4.5: Experimentos com diversidade no ambiente de execução.

4.3.2 Resultados e Análises.

A Tabela 4.5 apresenta os resultados para o sistema com diversidade nos ambientes de execução. Considerando aspectos de desempenho, o comportamento é semelhante aos experimentos anteriores, apresentando uma queda nos cenários com uso de diversidade. No entanto, quando consideramos aspectos de segurança, o sistema fica mais robusto visto que uma combinação de vulnerabilidades (e não apenas uma) é necessária ser explorada por um atacante para comprometer o sistema. Por fim, é importante acrescentar que neste caso, devido a utilização de máquinas com menor poder de processamento [39], o desempenho ficou reduzido em comparação aos experimentos anteriores.

4.4 Considerações

Pelos testes com a implementação vazia, conclui-se que a implementação apresentada neste trabalho possui *overhead* mínimo sobre a performance da aplicação. Porém, o sistema como um todo fica limitado pela performance da implementação mais lenta, portanto deve haver uma preocupação durante o desenvolvimento para obter a melhor performance em cada ambiente. Por exemplo, ao invés de utilizar a mesma implementação de lista, poderiam ser utilizadas estruturas específicas de cada linguagem, o que iria conferir uma melhor performance.

Capítulo 5

Conclusões

Este capítulo apresenta um sumário daquilo que foi desenvolvido, principais contribuições e perspectivas para continuidade do projeto.

5.1 Visão Geral do Trabalho

Este trabalho apresentou uma nova abordagem para RME que utiliza diversidade na implementação das réplicas para aumentar a segurança das aplicações. De fato, é muito provável que uma mesma vulnerabilidade não possa ser explorada em mais de uma linguagem de programação, i.e., cada linguagem e implementação apresentará suas próprias vulnerabilidades.

5.2 Revisão dos Objetivos e Contribuições

Cada um dos objetivos específicos deste trabalho foram alcançados conforme discutido a seguir:

1. **Estudar os conceitos envolvendo diversidade e RME.** Esse objetivo foi alcançado através de uma revisão de bibliografia e revisão de estado da arte, que embasam essa monografia, conforme discutido no Capítulo 2.

2. **Proposta de uma arquitetura para suportar diversidade na implementação de réplicas de uma RME.** Esse objetivo foi atingido através da arquitetura proposta na Seção 3.2, e devido à utilização de camadas intermediárias em C com utilização de JNI (Seção 2.4.1) e Protocol Buffers (Seção 2.4.2), é extensível e com boa performance, como evidenciado nos experimentos do Capítulo 4.

3. **Integração e descrição de como esta arquitetura foi integrada no Bft-SMaRt.** Foram implementadas integração com 3 linguagens de programação além do Java (C, C++ e Python), como descrito na Seção 3.2, e descrição dos passos para implementação de uma aplicação com diversidade na Seção 3.3.

4. **Apresentação e análise de experimentos com as implementações realizadas.** Por fim, este objetivo foi alcançado através dos experimentos descritos no Capítulo 4, concluindo que a implementação é viável e apresenta boa performance, analisando o comportamento do sistema quanto à diferença de performance das diferentes linguagens de programação, e observando como evitar problemas de performance decorrentes disso.

Todas as implementações desenvolvidas estão disponíveis como *open-source* no seguinte repositório: `https://github.com/caioycosta/bftsmart-diversity`.

A partir deste trabalho, foi produzido um artigo [15], o qual foi apresentado no XVII Workshop de Testes e Tolerância a Falhas (WTF 2016).

5.3 Perspectivas Futuras

Como trabalhos futuros, pretende-se implementar suporte para mais linguagens de programação, além de explorar outras aplicações com o intuito de aumentar a compreensão sobre o funcionamento de uma Replicação Máquina de Estados com diversidade. Além disso, também pretendemos analisar os efeitos da utilização de diversidade nos clientes, embora este não seja o foco principal do trabalho, e realizar mais experimentos variando principalmente a localização do líder (processo mais sobrecarregado durante a execução da RME, pois é responsável pela elaboração das propostas de ordenação do algoritmo de difusão atômica [8, 9]), analisando características de hardware e de outras linguagens.

Referências

[1] Harold Abelson and Gerald Jay Sussman. Structure and interpretation of computer programs. 1983. 24

[2] Eduardo Adilio Pelinson Alchieri, Alysson Neves Bessani, and Joni da Silva Fraga. Replicação máquina de estados dinâmica. In *Anais do XIV Workshop de Teste e Tolerância a Falhas- WTF 2013*, 2013. 7

[3] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, 2011. 1

[4] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, March 2004. 1

[5] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *International Computer Software and Applications Conference*, 1977. 9

[6] Eli Barzilay and Dmitry Orlovsky. Foreign interface for plt scheme. *on Scheme and Functional Programming*, page 63, 2004. 12

[7] Alysson Bessani, Alessandro Daidone, Ilir Gashi, Rafael Obelheiro, Paulo Sousa, and Vladimir Stankovic. Enhancing fault / intrusion tolerance through design and configuration diversity. In *Proceedings of the 3rd Workshop on Recent Advances on Intrusion-Tolerant Systems*, 2009. 1, 10

[8] Alysson Bessani, João Sousa, and Eduardo Alchieri. State machine replication for the masses with BFT-SMaRt. 2014. 1, 2, 7, 9, 17, 27, 42, 43, 48

[9] Miguel Castro and Barbara Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002. 1, 7, 48

[10] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. Base: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3):236–269, 2003. 2, 11

[11] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, pages 3–9, 1978. 10

[12] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riché. UpRight cluster services. In *Proc. of the ACM SOSP'09*, 2009. 1

[13] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 153–168. USENIX Association, 2009. 1

[14] Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In *Computer Safety, Reliability, and Security*, pages 235–248. Springer, 2003. 1, 6

[15] Caio Yuri Silva Costa e Eduardo Adilio Pelinson Alchieri. Implementando diversidade em replicação máquina de estados. In *Anais do XVII Workshop de Testes e Tolerância a Falhas–WTF 2016*, 2016. 48

[16] Python Software Foundation. ctypes—a foreign function library for python. https://docs.python.org/2/library/ctypes.html, 2015. Accessado: 2015-10-18. 14

[17] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. Os diversity for intrusion tolerance: Myth or reality? In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, 2011. 1, 10

[18] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience*, 44(6):735–770, 2014. 1, 10, 11

[19] Google. Protocol buffers developers guide. https://developers.google.com/protocol-buffers/docs/overview, 2015. Accessado: 2015-06-07. 15, 16

[20] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *Proceedings of the ACM SIGOPS/EuroSys European Systems Conference*, 2010. 1

[21] Vassos Hadzilacos and Sam Toueg. A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical report, Department of Computer Science, Cornell, May 1994. 6

[22] Mark Kenneth Joseph. Architectural issues in fault-tolerant, secure computing systems. Technical report, California Univ., Los Angeles, CA (USA), 1988. 10

[23] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39, December 2009. 1

[24] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982. vii, 5

[25] Jean-Claude Laprie. *Dependability: Basic concepts and terminology*. Springer, 1992. 4

[26] Siliang Li and Gang Tan. Exception analysis in the java native interface. *Science of Computer Programming*, 89:273–297, 2014. 12

[27] Rafael Obelheiro, Alysson Neves Bessani, and Lau Cheuk Lung. Analisando a viabilidade da implementaçao prática de sistemas tolerantes a intrusoes. *Anais do V Simpósio Brasileiro em Segurança da Informaç ao e de Sistemas Computacionais-SBSeg 2005*, 2005. 2, 11

[28] Rafael Rodrigues Obelheiro, Alysson Neves Bessani, and Lau Cheuk Lung. Analisando a viabilidade da implementação prática de sistemas tolerantes a intrusões. In *Anais do V Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg 2005*, 2005. 1, 10

[29] Oracle. Java native interface specification. `http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html`, 2014. Accessado: 2015-06-01. 13, 14

[30] Marco Platania, Daniel Obenshain, Thomas Tantillo, Ricky Sharma, and Yair Amir. Towards a practical survivable intrusion tolerant replication system. In *33rd IEEE International Symposium on Reliable Distributed Systems*, pages 242–252, 2014. 1, 10

[31] Java Native Access project. Jna api documentation—overview. `http://java-native-access.github.io/jna/4.2.1/`, 2015. Accessado: 2016-6-18. 12

[32] Java Native Access project. Java native access (jna)—readme. `https://github.com/java-native-access/jna/blob/master/README.md`, 2016. Accessado: 2016-6-18. 12

[33] B. Randell. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, 1975. 9

[34] Fred B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990. 1, 6, 7

[35] Beth Stearns. Java native interface. http://www.sc.ehu.es/towmogoj/tutorial%20Java/native1.1/, 2004. Accessado: 2015-04-09. 12, 13, 14

[36] Kurt Vanmechelen and Jan Broeckhove. Interfacing c++ member functions with c libraries. *Jack Dongarra University of Tennessee and Oak Ridge National Laboratory and Kaj Madsen and Jerzy Wasniewski*, page 204, 2004. 23

[37] Kenton Varda. Protocol buffers: Google's data interchange format. http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html, 2008. Accessado: 2015-04-07. 15

[38] Giuliana Veronese, Miguel Correia, Alysson Bessani, Lau Lung, and Paulo Verissimo. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 62(1), January 2013. 1

[39] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of 5th Symp. on Operating Systems Design and Implementations*, 2002. 41, 45, 46